Olomoucký informatický korespondenční seminář – Olinx

Programovací jazyk PROLOG, 2. díl

Miroslav Kolařík

olinx.inf.upol.cz

Úvod

V tomto dílu se budeme věnovat, v PROLOGu velmi často používaným strukturám, tak zvaným „seznamům“. Seznamy nejprve definujeme a seznámíme se s jejich značením. Poté si na konkrétních ukázkách představíme základní operace se seznamy a přiblížíme si tak práci s nimi. Následně se zaměříme na jednoduché aritmetické výpočty a vyhodnocování aritmetických výrazů. Na závěr jsou opět nachystány úkoly, za celkem 25 bodů, na kterých si můžete ověřit své dosavadní znalosti z tohoto semináře.

Seznamy a práce s nimi

Seznam je jednoduchá datová struktura široce používaná v nenumerickém programování. Seznamem může být posloupnost libovolného počtu položek, například jaro, léto, podzim, zima. Tento seznam může být v PROLOGu napsán takto:

```
[jaro,leto,podzim,zima]
```

Toto je ale pouze externí vzhled seznamu. Všechny struktury jsou v PROLOGu stromy[1] a seznamy nejsou výjimkou. Jak může být seznam reprezentován jako standardní objekt PROLOGu? Musíme vzít v úvahu dva případy: seznam je buď prázdný, nebo neprázdný. V prvním případě je seznam jednoduše zapsán takto: `[]`. Ve druhém případě seznam sestává z:

- první položky, nazývané hlava seznamu,
- zbývající části seznamu, nazývané ocas.

Pro náš ukázkový seznam

```
[jaro,leto,podzim,zima]
```

je hlavou `jaro` a ocasem seznam:

```
[leto,podzim,zima]
```

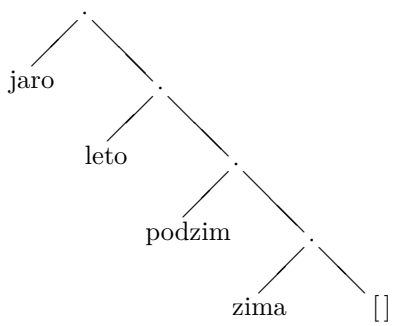
Obecně, hlavou seznamu může být cokoliv (libovolný PROLOGovský objekt, například strom, nebo proměnná); ocas musí být vždy seznamem. Hlava a ocas jsou pak spojeny do struktury speciálním funktorem,

```
.(Hlava, Ocas)
```

Jelikož je ocas opět seznamem, je buď prázdný, nebo má vlastní hlavu a ocas. Proto k reprezentaci seznamu libovolné délky nepotřebujeme žádný další princip. Náš ukázkový seznam je tedy reprezentován jako:

```
.(jaro,.(leto,.(podzim,.(zima,[]))))
```

a následující obrázek ukazuje odpovídající stromovou strukturu.



My budeme nadále používat zejména reprezentaci seznamu pomocí hranatých závorek, přičemž budeme mít na paměti, že interní reprezentace je realizována pomocí binárních[2] stromů a odpovídá zápisu pomocí teček (tzv. tečkových párů). Poznamenejme ještě, že prvky seznamu mohou být objekty libovolného druhu; tedy i opět seznamy, například

```
[ester,[19,12],simon,[22,12]]
```

Uvedený seznam má čtyři prvky, kde druhým a čtvrtým prvkem je dvouprvkový seznam čísel.

Často je praktické mít možnost zacházet s celým ocasem jako s jediným objektem. Například, nechť

```
L=[a,b,c]
```

Pak můžeme psát `Ocas=[b,c]` a `L=.(a,Ocas)`.

Toto lze v notaci hranatých závorek pro seznamy vyjádřit pomocí svislé čárky (která separuje hlavu a ocas) takto:

```
L=[a|Ocas]
```

Zápis pomocí svislé čárky je ve skutečnosti ještě obecnější. Můžeme uvést libovolný počet prvků, po kterých následuje „`|`“ a seznam zbývajících položek. Alternativní zápisy výše uvedeného seznamu jsou:

```
[a,b,c]=[a|[b,c]]=[a,b|[c]]=[a,b,c|[]]
```

Vybrané operace na seznamech

První se zaměříme na operaci náležení (membership), ověřující, zda je nějaký objekt prvkem seznamu. Implementaci provedeme s binární relací

```
member(X,L)
```

kde `X` je objekt a `L` je seznam. Dotaz `member(X,L)` je pravdivý, jestliže se `X` vyskytuje v `L`. Například

```
member(b,[a,b,c])
```

je pravda,

```
member(b,[a,[b,c]])
```

není pravda a

[1] Pod pojmem strom zde máme na mysli souvislý, neorientovaný graf bez kružnic.

[2] U binárních stromů má každý vrchol nejvýše dva potomky.

```
member([b,c],[a,[b,c]])
```

pravda je. Program pro relaci náležení vychází z následujícího pozorování. `X` je prvkem `L`, jestliže

(1) `X` je hlavou `L`, nebo
(2) `X` je prvkem ocasu `L`.

Oba dva body zapíšeme pomocí dvou klauzulí; první je jednoduchý fakt a druhá klauzule je pravidlo:

```
member(X,[X,Ocas]).
member(X,[Hlava|Ocas]) :- member(X,Ocas).
```

Druhou operací se seznamy, kterou se budeme zabývat je spojení (concatenation) dvou seznamů do třetího seznamu. Pro spojení (též zřetězení) seznamů zavedeme ternární relaci:

```
conc(L1,L2,L3)
```

Zde jsou `L1` a `L2` dva seznamy a `L3` jejich spojení. Například

```
conc([a,b],[c,d],[a,b,c,d])
```

je pravda, naproti tomu

```
conc([a,b],[c,d],[a,b,a,c,d])
```

je nepravda. Při definování relace `conc` budeme mít opět dva případy, závislé na prvním argumentu `L1`:

(1) Jestliže je prvním argumentem prázdný seznam, pak musí být druhý a třetí argument stejný seznam `L`.
(2) Jestliže první argument relace `conc` je neprázdný seznam, pak musí mít hlavu a ocas a musí vypadat takto:

    ```
    [X|L1]
    ```

    Výsledkem spojení seznamu `[X|L1]` a nějakého seznamu `L2` je seznam `[X|L3]`, kde `L3` je spojením seznamů `L1` a `L2`.

Oba dva body v PROLOGu vyjádříme jako jeden fakt a jedno pravidlo:

```
conc([],L,L).

conc([X|L1],L2,[X|L3]) :- conc(L1,L2,L3).
```

Tento program již můžeme použít na spojení dvou daných seznamů, například:

```
?- conc([a,b,c],[1,2,3],L).

L=[a,b,c,1,2,3].
```

```
?- conc([a,[b,c],d],[a,[],b],L).

L=[a,[b,c],d,a,[],b].
```

Přestože program `conc` vypadá poněkud jednoduše, může být flexibilně využit i jinými způsoby. Například můžeme `conc` využít obráceným způsobem pro rozklad daného seznamu na dva seznamy, například takto:

```
?- conc(L1,L2,[a,b,c]).

L1=[]
L2=[a,b,c];

L1=[a]
L2=[b,c];

L1=[a,b]
L2=[c];

L1=[a,b,c]
L2=[].
```

Je tedy možné seznam `[a,b,c]` rozložit čtyřmi různými způsoby, které všechny nalezl náš program `conc` přes backtracking.

Program lze využít i jinak. Například můžeme najít dny, které předcházejí a dny, které následují za daným dnem v týdnu, jako v tomto dotazu:

```
?- conc(Pred,[st|Po],[po,ut,st,ct,pa,so,ne]).

Pred=[po,ut]
Po=[ct,pa,so,ne].
```

Můžeme také zjistit bezprostředního předchůdce a bezprostředního následovníka středy přes dotaz:

```
?- conc(_,[Den1,st,Den2|_],[po,ut,st,ct,pa,so,ne]).

Den1=ut
Den2=ct.
```

Můžeme ještě vymazat z nějakého seznamu `L1` vše, co následuje za třemi bezprostředními výskyty `z` v `L1` spolu s těmi třemi `z`. Například:

```
?- L1=[a,b,z,z,c,z,z,z,d,e], conc(L2,[z,z,z|_],L1).

L1=[a,b,z,z,c,z,z,z,d,e]
L2=[a,b,z,z,c].
```

Už jsme naprogramovali relaci náležení: `member`. S využitím relace `conc` ji lze elegantně naprogramovat takto:

```
member1(X,L) :- conc(L1,[X|L2],L).
```

Toto pravidlo říká: `X` je prvkem seznamu `L`, jestliže může být seznam `L` rozložen na dva seznamy tak, že druhý seznam má `X` jako hlavu. Pravidlo `member1`[3] může být zapsáno s využitím anonymních proměnných:

```
member1(X,L) :- conc(_,[X|_],L).
```

Nyní se zaměříme na to, jak přidat (add) do seznamu položku na první místo. Nová položka se tak stane novou hlavou seznamu. Jestliže `X` je nová položka a `L` je seznam, ke kterému má být `X` přidáno, pak má výsledný seznam tvar:

```
[X|L]
```

Nepotřebujeme tedy proceduru pro přidávání nového prvku na začátek seznamu. Nicméně, kdybychom ji chtěli explicitně definovat, tak ji můžeme vytvořit jako následující fakt:

```
add(X,L,[X|L]).
```

[3] Samozřejmě, `member1` definuje stejnou relaci jako `member`. Použili jsme různá jména jen kvůli rozlišení dvou různých implementací.

Dále se budeme zabývat vymazáním (delete) prvku `X` ze seznamu `L`. Použijeme k tomu relaci

```
del(X,L,L1)
```

kde `L1` je rovno seznamu `L` bez prvku `X`. Relace `del` bude definována podobně jako relace `member`. Opět máme dva případy:

(1) Jestliže `X` je hlavou seznamu, pak výsledkem po smazání je ocas seznamu.
(2) Jestliže se prvek `X` vyskytuje v ocasu seznamu, pak je odtud smazán.

```
del(X,[X|Ocas],Ocas).
del(X,[Y|Ocas],[Y|Ocas1]) :- del(X,Ocas,Ocas1).
```

Podobně jako u výše zavedené operace náležení (`member`) je relace `del` nedeterministická. Pokud má `X` více výskytů v seznamu, pak je `del` schopna smazat každý z nich pomocí backtrackingu. Samozřejmě každé alternativní provedení smaže pouze jeden výskyt `X`, přičemž ostatní výskyty zůstanou zachovány. Například:

```
?- del(a,[a,b,a,a],L).

L=[b,a,a];
L=[a,b,a];
L=[a,b,a].
```

Relace `del` neuspěje, pokud seznam neobsahuje položku, která má být smazána, například:

```
?- del(x,[a,b,c,d],List).

false.
```

Relace `del` může být použita také jiným způsobem. Například, pokud chceme vložit prvek `a` do seznamu `[1,2,3]`, můžeme to udělat tak, že položíme dotaz: co je `L`, když po vymazání prvku `a` z `L` získáme seznam `[1,2,3]`?

```
?- del(a,L,[1,2,3]).

L=[a,1,2,3];
L=[1,a,2,3];
L=[1,2,a,3];
L=[1,2,3,a].
```

Operaci vložení (insert) prvku `X` na libovolné místo seznamu `Seznam` (dávající `VetsiSeznam`), lze zavést pravidlem:

```
insert(X,Seznam,VetsiSeznam) :-
  del(X,VetsiSeznam,Seznam).
```

V relaci `member1` jsme, pomocí relace `conc`, elegantně implementovali relaci náležení. Můžeme k tomu využít i relaci `del`. Idea je jednoduchá: `X` je členem seznamu `Seznam`, jestliže lze `X` z tohoto seznamu smazat.

```
member2(X,Seznam) :-
  del(X,Seznam,_).
```

Na závěr této části se budeme zabývat podseznamy. Budeme uvažovat relaci `podseznam`, která bude mít dva argumenty: seznam `L` a seznam `S`, přičemž seznam `S` bude podseznamem seznamu `L`. Chceme, aby odpověď na dotaz

```
?- podseznam([c,d,e],[a,b,c,d,e,f]).
```

byla pravda (`true`), ale odpověď na dotaz

```
?- podseznam([c,e],[a,b,c,d,e,f]).
```

byla nepravda (`false`).

PROLOGovský program pro relaci `podseznam` je založen na stejné myšlence jako program pro relaci `member1`, pouze s tím rozdílem, že relace `podseznam` je obecnější. V souladu s tím ji můžeme vytvořit takto:
`S` je podseznam `L`, jestliže:

(1) `L` může být rozložen do dvou seznamů: `L1` a `L2`,
(2) `L2` může být rozložen do dvou seznamů: `S` a `L3`.

Jak jsme již uvedli, relace `conc` může být použita pro rozklad seznamů. Tedy odpovídající program v PROLOGu může být napsán takto:

```
podseznam(S,L) :-
  conc(L1,L2,L),
  conc(S,L3,L2).
```

Relaci `podseznam` lze samozřejmě flexibilně několika způsoby využít. Ačkoli byla relace navržena pro ověření, zda se nějaký seznam vyskytuje v daném seznamu jako podseznam, může být použita, například, pro nalezení všech podseznamů daného seznamu:

```
?- podseznam(S,[a,b,c]).

S=[];
S=[a];
S=[a,b];
S=[a,b,c];
S=[];
S=[b];
...
```

Permutace

Občas je užitečné vygenerovat všechny permutace daného seznamu. Za tímto účelem definujeme relaci `perm` se dvěma argumenty. Argumenty jsou dva seznamy, takové, že jeden je permutací druhého. Záměrem je vytvářet permutace seznamu pomocí backtrackingu, jak je naznačeno v následujícím příkladu:

```
?- perm([a,b,c],P).

P=[a,b,c];
P=[a,c,b];
P=[b,a,c];
...
```

Program pro generování permutací bude opět založen na uvážení dvou případů souvisejících s prvním seznamem:

(1) Jestliže je první seznam prázdný, musí být i druhý seznam prázdný.
(2) Je-li první seznam neprázdný, tedy je ve tvaru `[X|L]`, pak nejprve vytvoříme permutace seznamu `L`, čímž obdržíme seznam `L1`, do kterého poté stačí vložit prvek `X` na každou možnou pozici.

Těmto dvěma případům odpovídají tyto dvě PROLOGovské klauzule:

```
perm([],[]).
perm([X|L],P) :-
  perm(L,L1), insert(X,L1,P).
```

Jednou z alternativ k tomuto programu by bylo smazat prvek `X` z prvního seznamu, permutovat zbytek a obdržet tak seznam `P` a pak přidat `X` na začátek `P`. Odpovídající program následuje:

```
perm2([],[]).
perm2(L,[X|P]) :-
  del(X,L,L1), perm2(L1,P).
```

Při běžném použití fungují obě uvedené relace (`perm` a `perm2`) korektně a dávají očekávané výsledky. Například pro dotaz

```
?- perm([cervena,modra,bila],P).
```

dostaneme jako výsledek všech šest permutací:

```
P=[cervena,modra,bila];
P=[cervena,bila,modra];
P=[modra,cervena,bila];
P=[modra,bila,cervena];
P=[bila,cervena,modra];
P=[bila,modra,cervena].
```

Pokud ale zadáme dotaz ve tvaru

```
?- perm(L,[a,b,c]).
```

dostane se program do nekonečné smyčky, podobně jako relace `perm2`. Vyzkoušejte si to. Je proto na místě nezbytná obezřetnost při jejich používání.

Základy aritmetiky

V této části se krátce zaměříme na základní aritmetické operace a jejich vyhodnocování. Nejprve si představíme některé předdefinované operátory:

`+`	sčítání
`-`	odčítání
`*`	násobení
`/`	dělení
`**`	mocnina
`//`	celočíselné dělení
`mod`	modulo, zbytek po celočíselném dělení.

Jedná se o výjimečný případ, kdy se operátor může chovat jako operace. V takových případech je však nutná další indikace k provádění aritmetických výpočtů. Následující dotaz je naivním pokusem o aritmetický výpočet:

```
?- X=1+2.
```

PROLOG totiž odpoví

```
X=1+2.
```

a ne `X=3`, jak jsme nejspíše očekávali. Důvod je jednoduchý: výraz `1+2` pouze označuje PROLOGovský term, kde `+` je funktor a `1` a `2` jsou jeho argumenty. V posledním dotazu není nic, co by iniciovalo výpočet (aktivovalo operaci sčítání). Tento problém řeší předdefinovaný operátor `is`. Právě operátor `is` vynucuje vyhodnocení. Správný způsob k vyvolání vyhodnocení aritmetického výrazu je:

```
?- X is 1+2.
```

Nyní bude odpověď

```
X=3.
```

Sčítání zde bylo provedeno zvláštním postupem, pomocí speciální procedury[4], která je spojena (asociována) s operátorem `is`. Podobně fungují i další výše uvedené předdefinované operátory. Vyzkoušejme některé položením složeného dotazu:

```
?- X is 5/2, Y is 5//2, Z is 5 mod 2.

X=2.5,
Y=2,
Z=1.
```

Levým argumentem operátoru `is` je jednoduchý objekt. Pravým argumentem je aritmetický výraz složený z aritmetických operátorů, čísel a proměnných (jejichž hodnoty musí být při provádění výpočtu známy).

V PROLOGu také můžeme porovnávat aritmetické výrazy, jako například, zda-li je součin čísel 277 a 37 větší než 10000:

```
?- 277*37>10000.

true.
```

Poznamenejme, že podobně jako `is`, operátor „`>`“ vynutí vyhodnocování.

Představme si operátory pro porovnávání:

`X>Y`	`X` je větší než `Y`
`X<Y`	`X` je menší než `Y`
`X>=Y`	`X` je větší nebo rovno než `Y`
`X<=Y`	`X` je menší nebo rovno než `Y`
`X=:=Y`	hodnoty `X` a `Y` jsou stejné
`X=\=Y`	hodnoty `X` a `Y` nejsou stejné.

Dodejme ještě, že operátor „`=`“ se podstatně liší od operátoru „`=:=`“. Rozdíly si nejprve demonstrujme na konkrétních příkladech:

```
?- 1+2=:=2+1.

true.

?- 1+2=2+1.

false.

?- 1+A=B+2.

A=2,
B=1.
```

Například, máme-li dotazy ve tvaru „`X=Y`“ a „`X=:=Y`“, pak první z nich způsobuje porovnání objektů `X` a `Y` a může (pokud se `X` a `Y` shodují) konkretizovat hodnoty proměnných, které se v nich vyskytují. Na druhou stranu druhý operátor „`=:=`“ způsobuje aritmetické vyhodnocení a nemůže být nikdy použit ke konkretizaci hodnot proměnných.

[4] Takové procedury nazýváme vestavěné, anglicky built-in procedures.

Dále si ukážeme dva příklady, na kterých demonstrujeme použití aritmetických operací. První bude výpočet největšího společného dělitele, zatímco druhý bude sloužit k určení počtu položek daného seznamu.

Mějme dána dvě přirozená čísla `X` a `Y`. Jejich největší společný dělitel `D` určíme na základě následujících tří případů:

(1) Jestliže se `X` a `Y` rovnají, pak je `D` roven `X`.
(2) Jestliže je `X<Y`, pak je `D` roven největšímu společnému děliteli `X` a rozdílu `Y-X`.
(3) Jestliže je `X>Y`, pak postupujeme jako v případě (2), s tím, že vyměníme `X` a `Y`.

Tato pravidla přeformulujeme do PROLOGovského programu definováním ternární relace `nsd(X,Y,D)`. Dostaneme tak následující program:

```
nsd(X,X,X).

nsd(X,Y,D) :-
  X<Y,
  Y1 is Y-X,
  nsd(X,Y1,D).

nsd(X,Y,D) :-
  Y<X,
  nsd(Y,X,D).
```

Samozřejmě by poslední řádek v programu mohl být nahrazen těmito dvěmi řádky:

```
  X1 is X-Y,
  nsd(X1,Y,D).
```

Funkčnost programu otestujte na několika konkrétních příkladech.

Druhým příkladem je určení délky seznamu. I zde se nám bude hodit aritmetika. Budeme totiž počítat počet položek v seznamu. K tomuto účelu definujme proceduru `delka(Seznam,N)` se dvěma argumenty, která bude počítat prvky v seznamu `Seznam` a v `N` zaznamenávat jejich počet. Jistě bude užitečné uvažovat dva případy:

(1) Jestliže je seznam prázdný, pak je jeho délka `0`.
(2) Pokud seznam není prázdný, tedy je ve tvaru `Seznam=[Hlava|Ocas]`, pak je jeho délka rovna číslu `1` plus délka ocasu `Ocas`.

Tyto dva případy korespondují s následujícím programem:

```
delka([],0).

delka([_|Ocas],N) :-
  delka(Ocas,N1),
  N is 1+N1.
```

Program je hotov. Zkuste si sami rozmyslet, co by se stalo, kdybychom v něm prohodili poslední dva řádky. My si program vyzkoušíme položením jednoho konkrétního dotazu:

```
?- delka([a,b,[c,d],e],N).

N=4.
```

Závěrem

Seznámili jsme se se seznamy, v PROLOGu často používanými datovými strukturami. Víme, že seznam je buď prázdný, nebo sestává z „hlavy“ a „ocasu“, který je také seznamem. V souvislosti s operacemi se seznamy, umíme naprogramovat relaci pro: náležení do seznamu, spojení dvou seznamů, přidání prvku do seznamu, smazání prvku ze seznamu, podseznam.

Dozvěděli jsme se, že aritmetika je v PROLOGu spjata s vestavěnými procedurami. Vyhodnocení[5] aritmetického výrazu je zajištěno přes proceduru `is` a také pomocí operátorů pro porovnávání: `<`, `<=` a tak dále. Poznali jsme také vestavěné procedury asociované s předdefinovanými operátory: `+`, `-`, `*`, `/` a podobně.

Úkoly k odevzdávání

Následují úkoly, které se odevzdávají přes web semináře Olinx,

https://olinx.inf.upol.cz/

Úkoly stačí odevzdat ve formě jednoho jednoduchého textového souboru, ve kterém jsou napsány řešení ke všem úkolům. Pro tvorbu řešení se předpokládá používání aplikace SWI-PROLOG. U všech úkolů předpokládejte korektní vstupy.

Úkol 1 *5 bodů*

Definujte dvě unární relace `sudadelka(Seznam)` a `lichadelka(Seznam)`, které jsou pravdivé, pokud je jejich argumentem seznam se sudým, respektive lichým, počtem prvků. Pak tedy například:

```
?- sudadelka([a,b,c,d]).

true.

?- lichadelka([a,b,c,d]).

false.
```

Úkol 2 *8 bodů*

Definujte relaci `obratit(Seznam,ObracenySeznam)`, která obrací seznamy tak, aby například:

```
?- obratit([a,b,c,d],S).

S=[d,c,b,a].
```

Dále definujte unární relaci `palindrom(Seznam)`. Poznamenejme, že seznam je palindromem, pokud je při čtení zepředu stejný jako při čtení odzadu. Tedy například:

```
?- palindrom([m,a,d,a,m]).

true.

?- palindrom([d,a,m,a]).

false.
```

[5] Při vyhodnocování aritmetického výrazu musí mít všechny argumenty číselné hodnoty.

Úkol 3 *6 bodů*

Napište PROLOGovský program pro ternární relaci `generujseznam(N1,N2,Seznam)`, který bude sloužit k vygenerování celočíselného seznamu `Seznam` od hodnoty `N1` do hodnoty `N2`. Pak tedy například:

```
?- generujseznam(3,7,Seznam).

Seznam=[3,4,5,6,7].
```

Úkol 4 *6 bodů*

Napište PROLOGovský program pro binární relaci `soucetseznamu(Seznam,Soucet)` tak, aby byl `Soucet` součtem hodnot daného číselného seznamu `Seznam`. Pak tedy například:

```
?- soucetseznamu([2,2,6,5,1],Soucet).

Soucet=16.
```

Literatura

[1] I. Bratko, 2011. *PROLOG Programming for Artificial Intelligence (4th Edition)*. Pearson Education Canada. ISBN 9780321417466.